Algorithms

Eyal Arubas

February 8, 2013

This book is based on an algorithms course I took in 2012 at the Hebrew University of Jerusalem, Israel. The material is based on my notes from the lectures of Prof. Alex Samorodnitsky, as well as my readings of Introduction To Algorithms (T.H. Corman), Algorithms (U.V. Vazirani), entries in Wikipedia and more.

I wrote this book because I find the subject interesting, and it helped me prepare for my exam. Hopefully it will help whoever is reading it as well.

Needless to say, I take no responsibility for the accuracy, completeness and correctness of what is written here. I'm not, in any way, an authority on algorithms, so take it as it is. That being said, I still wish this book to be as helpful as possible, so if you find any mistakes or inaccuracies, please send me an email to **EyalArubas@gmail.com**.

Also, you are more than welcomed to just tell me what you think of this book. I like the feedback.

The structure of the book is such that each chapter will begin with several examples of relevant problems. These examples will demonstrate a situation in which we need to reach a solution by solving a problem of a certain type. Then in the rest of the chapter we will discuss methods for solving this type of problems. Each chapter is independent, so you can just jump directly to a subject of your choice.

The latest version of this book can be downloaded from my website at **http://EyalArubas.com**.

I encourage you to share this book and pass it along, if you find it useful of course.

Contents

Transversality is the characterization of intersection between sets.

Chapter 1

Greedy Algorithms

1.1 Motivation

1.1.1 Robbing a Bank

Let's say we are bank robbers, and we want to maximize our profit. The problem is that we brought just one bag, which can contain up to a 100 units of stolen items. We rob the bank at night, so it's empty of people, and we have enough time to think which items we want to take.

The inventory of the bank is as follows:

Item	Amount	Total Worth
Gold	10	1000
Silver	50	1500
Bronze	100	2000

As the robbers, our task is to figure out how many units of Gold, Silver and Bronze we put in the bag, such that our profit is maximal. In other words, if we take x units of Gold, y units of Silver and z units of Bronze, such that $x + y + z = 100$, what would be the best choice of x, y and z?

For example:

- If we take just 100 units of bronze ($x = 0, y = 0, z = 100$), our profit would be 2000.
- If we take 50 units of Bronze and 50 units of Silver ($x = 0, y = 50, z = 50$), our profit would be 2500.
- If we take 90 units of Bronze and 10 units of Silver ($x = 10, y = 0, z = 90$), our profit would be 2800.

Since we are robbers, we are naturally greedy, so we decide to take as much as possible from the more expensive items first. We make the following choice:

- 10 units of Gold, 50 units of Silver, and fill the rest of the bag with the least expensive item - 40 units of Bronze ($x = 10, y = 50, z = 40$). In that case our profit is 3300.

As we will later learn to prove, this is the maximal possible profit.

1.2 Introduction

As we have seen, certain problems can be solved by employing a greedy strategy. By considering what is best **now**, without regarding the consequences **later**, we can reach an optimal solution. Obviously not all problems can be solved optimally by this approach. For most problems, we can't just ignore the consequences of our current choices and we must look at the problem with a global view. So in order to justify the use of the greedy approach, we want to be able to identify the problems for which an optimal solution is guarantied by this approach.

Apparantly, all problems which can be solved greedily exhibit certain mathematical characteristics. We call the mathematical entities which adhere to these characteristics "Matroids".

The characteristics of matroids make them very relevant to optimization problems, which can be solved with greedy algorithms. We will later see that by taking an optimization problem and finding a matroid which describes it, we can solve it optimally with a generic greedy algorithm. The challange, for most problems, will be to formulate the problem in such a way that it will correspond to a matroid.

Greedy algorithms are a somewhat unique family of algorithms, in the sense that:

1. They solve certain optimization problems. Meaning, by using the greedy algorithm, we can find the optimal solution to the problem.

2. We can prove it.

The fact that we can prove that greedy algorithms solve certain optimization problems, is not trivial, and is thanks to matroids.

But before discussing Matroids, we first show several optimization problems and solve them with *exchange lemmas*. Then a mathematical discussion about matroids will be presented, and will seem a bit unrelated to algorithms. After we complete this discussion, another discussion will be made about the equivalency of matroids to optimization problems, which will be demonstrated through more examples. In the last part of the chapter we will see the full process of defining a problem, transforming it into a matroid and finding an optimal solution through a generic greedy algorithm.

1.3 Exchange Lemmas

Oftentimes we can formulate an algorithm which we theorize solves some optimization problem. If it indeed does solve the problem, we want to be able to prove it. Usually we do it by saying that if some optimal solution to the problem exists, then it will coincide with the solution of our algorithm. Our goal, then, is to show that we can take the (currently unknown) optimal solution (which is usually not unique), manipulate it without damaging its optimality, and reach the solution given by our algorithm; thus showing that it, too, is an optimal solution.

For this purpose we use *exchange lemmas.* What these lemmas actually do is show that we can manipulate the uknown optimal solution, i.e. *exchange* part of it with part of our own solution, and still remain with an optimal solution. If we can show that eventually we can replace the **entire** uknown optimal solution with our own solution, and still remain optimal, then we have shown that our solution is as good and as optimal.

Unknown optimal solution sounds like a fallacy. How can a solution be optimal without knowing what it is? The fact is that we don't need to know exactly what this optimal solution is, but we just need to know what characteristics it must hold in order to be optimal.

To demonstrate, we show several examples.

1.3.1 Fractional Knapsack Problem

This is a similar problem to the "Robbing a bank" example given at 1.1.1. We have a knapsack which can carry a certain amount of weight. We also have a list of items; each item has it's own weight and value. We want to insert items into the knapsack such that the total value of items inside the knapsack is maximal. Notice that items needn't be whole, and can be inserted partially into the knapsack.

Our input is:

1. W - The maximum wight the knapsack can carry.
2. A list of n items. Item i is represented by the pair (v_i, w_i), where v_i is the value of the item and w_i is the weight of the item. All values and wights are non negative.

Our output should be:

A list of numbers $x_1, x_2, ..., x_n$, where x_i is the fractional amount of item i which is inserted into the knapsack ($0 \leq x_i \leq 1$).

The numbers x_i adhere to the constraint $\sum_{i=1}^{n} x_i w_i \leq W$.

Our goal is to maximize the total value of items in the knapsack $\sum_{i=1}^{n} x_i v_i$.

1.3.1.1 Algorithm

We propose a greedy algorithm which yields an optimal solution to this problem. We notice that greediness is the most natural approach in this case, since our

goal is to maximal the value of the knapsack, and we can (intuitivly) achieve that by grabing as much as possible from the most valued items.

1.3.2 Independent Vectors Set Problem

Suppose we have a finite vectors set F of n vectors, in some vector space V, and a positive weight functions on these vectors $\mu : V \to R^+$ (this function assigns a positive scalar value to each vector).

Our goal is to find a subset S of F ($S \subseteq F$), such that the vectors in S are linearly independent and the total weight of S is maximized ($\mu(S) = \sum_{v \in S} \mu(v)$).

1.3.2.1 Algorithm

In this problem, too, it's evident that greediness is the most intuitive approach. We want to maximize the weight of S, so we just add the vectors with the maximal weight, as long as linear independence in S is preserved.

Formally, our algorithm is:

Algorithm 1.1 Maximal weight independent vectors set algorithm

1. Sort the vectors in F by thier weight in descending order, such that $\mu(v_1) \geq \mu(v_2) \geq ... \geq \mu(v_n)$
2. Initialize S as the empty set: $S = \emptyset$
3. For $i = 1...n$:
   (a) If $S \cup \{v_i\}$ is linearly independent, update: $S = S \cup \{v_i\}$
4. Return S

1.3.2.2 Proof

We want to prove that algorithm 1.1 indeed returns a set of linearly independent vectors with maximal weight.

For this, we need to use the following lemma:

Lemma 1. *Let A, B two finite subsets of linearly independent vectors in vector space V. Suppose $|A| > |B|$. Then there is a vector $v \in A \backslash B$ such that $B \cup \{v\}$ in linearly independent.*

In others words, if A has more vectors than B, then we can find some vector v in A and add it to B such that $B \cup \{v\}$ is also linearly independent.

Proof. We need to show that there is a $v \in A \backslash B$ that is linearly independent with B. We will prove by negation. Let's suppose that there is no such vector. This means that **all** vectors in A are linearly dependent with B. In other

words, $\forall v \in A : v \in B$, which means that $span(A) \subseteq span(B)$. But then $\dim(span(A)) \leq \dim(span(B))$.

Because A and B are sets of linearly independent vectors, then $|A| = \dim(span(A))$ and $|B| = \dim(span(B))$, but this means that $|A| \leq |B|$, which is a contradiction to our assumption that $|A| > |B|$. Thus we conclude that indeed there is a vector $v \in A \backslash B$ such that $B \cup \{v\}$ in linearly independent. □

Now we can continue with the proof of the optimality of algorithm 1.1.

Remember that we want to prove that the set S which is returned by our algorithm has the maximal weight of all linearly independent subsets of F.

We will prove by negation.

Suppose there is some better, optimal, solution T such that $\mu(T) > \mu(S)$.

By lemma 1, $|S| = |T|$, because:

1. If $|S| > |T|$, then by the lemma, there is a vector $v \in S \backslash T$ such that $T \cup \{v\}$ is linearly independent. But this contradict the optimality of T, thus it's impossible.

2. If $|T| > |S|$, then by the lemma, there is a vector $v \in T \backslash S$ such that $S \cup \{v\}$ is linearly independent. But this contradicts the operation of our algorithm, which was supposed to add this vector v to S. So this is also impossible.

Thus indeed $|S| = |T|$.

Let's write the vectors in S and T by descending weight order:

$S = \{v_1, v_2, ..., v_k\} \quad \mu(v_1) \geq \mu(v_2) \geq ... \geq \mu(v_k)$

$T = \{u_1, u_2, ..., u_k\} \quad \mu(u_1) \geq \mu(u_2) \geq ... \geq \mu(u_k)$

Because $\mu(T) > \mu(S)$, then there must be some index i which is the first occurance of $\mu(u_i) > \mu(v_i)$.

We denote:

$A = \{v_1, ..., v_{i-1}\}$

$B = \{u_1, ..., u_{i-1}, u_i\}$

Obviously both A and B are sets of linearly independent vectors, and $|B| > |A|$. By lemma 1, there is a vector $u \in B \backslash A$ such that $A \cup \{u\}$ is linearly independent. Because S and T are ordered by descending weights, then the weight of this vector u (whichever it may be) is at least as the smallest-weight vector in B, which is u_i. In other words $\mu(u) \geq \mu(u_i)$. And by the definition of i, also $\mu(u_i) > \mu(v_i)$, and thus $\mu(u) > \mu(v_i)$.

Finally we notice that this contradicts the operation of our algorithm. Because, if $\mu(u) > \mu(v_i)$, then our algorithm should've chosen u before v_i (as we have seen, $A \cup \{u\}$ is linearly independent).

We have reached a contradiction, which finishes our proof. ■

1.4 Matroids

1.4.1 Properties of a Matroid

We define matroids through their properties. A matroid M is a pair of two entities:

1. A set, S, of elements.
2. A set, I, of subsets of S. More specifically, I is a subset of the power set* of S: $I \subseteq P(S)$.

We denote the matroid which is composed of S and I with $M = (S, I)$. We require I to hold three properties:

1. Must contain the empty set ($\emptyset \in I$).
2. Must hold the *Hereditary Property.*
3. Must hold the *Augmentation Property.*

1.4.1.1 Heriditary Property

We remember that I is a set of subsets of S. In other words, the elements of I are sets themselfs. If we take some set A in I ($A \in I$), then we can also look at subsets of A. Let's look at some subset of A and denote it with B, so we have $B \subseteq A$. Now we can ask an interesting question - does $B \in I$ hold? In other words, is B also an element in I? The answer is that sometimes it is and sometimes it isn't. But if it is, for all such A's and B's, then we say that I is *heriditary.*

Formal Definition I is heriditary if and only if $\forall A \in I, \forall B \subseteq A : B \in I$.

And in words - For every set A in I, every subset of A is also in I.

1.4.1.2 Augmentation Property

Suppose we have two subsets of S which are in I: $A, B \in I$. And suppose that B contains more elements than A. Then is it possible to take some element from B, which is not also in A, move it to A (**augment** A), and still remain in I? If it's possible for all A's and B's in I, then we say that I has the augmentation property.

Formal Definition I has the augmentation property if and only if $\forall A, B \in I, |B| > |A|, \exists x \in B \backslash A : A \cup \{x\} \in I$.

And in words - For all sets A and B in I, such that B is bigger than A, there exists an element x which is in B but not in A, such that the union of A and x is also in I.

*The power set, $P(S)$, of a set S, is defined to be the set of all the subsets of S. The power set is usually denoted by $P(S)$ or by 2^S. For example, given a set $S = \{1, 2\}$, then $P(S) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

Note: We only have to find just one such x.

1.4.2 The Generic Greedy Algorithm

As was mentioned in the introduction, a generic greedy algorithm will solve an optimization problem if it is defined as a matroid.

1.4.3 Examples

1.4.3.1 The Transversal Matroid

Transversality is the characterization of intersection between sets.

Let $C_1, C_2, ..., C_k$ be k sets of integers which comprise a disjoint union of $[n]$*: $C_1 \uplus C_2 \uplus ...C_k = [n]$.

We denote $S = [n]$, and we define $I = \{A \subseteq S : \forall i = 1, ..., k \, |A \cap C_i| \leq c\}$. I Contains all the subsets of S that have an intersection with each C_i of at most c elements.

Claim: $M = (S, I)$ is a matroid.

Proof: We need to prove that I holds each of the required three properties.

Contains the empty set: Suppose $A = \emptyset$. Obviously $A \subset S$ and for each $i = 1, ..., k$ it is evident that $|A \cap C_i| = |\emptyset \cap C_i| = 0 \leq c$. Thus $A \in I$. ■

Hereditary Property: We need to prove that $\forall A \in I, \forall B \subseteq A : B \in I$. Suppose $A \in I$ and $B \subseteq A$.

We need to show that $|B \cap C_i| \leq c$.

We know that $B \subseteq A$, so obviously $|B| \leq |A|$, thus it is evident that $|B \cap C_i| \leq |A \cap C_i|$ for every i. But because $A \in I$, we have $|A \cap C_i| \leq c$, which means $|B \cap C_i| \leq |A \cap C_i| \leq c$ for every i. Thus $B \in I$. ■

Augmentation Property: We need to prove that $\forall A, B \in I, |B| > |A| \, , \exists x \in B \backslash A : A \cup \{x\} \in I$.

Suppose $A, B \in I$ and $|B| > |A|$. We need to show that there is $x \in B \backslash A$ such that $A \cup \{x\} \in I$.

*$[n] = \{1, 2, 3, ..., n\}$

Chapter 2

Dynamic Algorithms

Chapter 3

Approximation Algorithms

3.1 Examples

3.1.1 Parallel Machine Online Scheduling

We have tasks we need to perform, and several machines to perform those tasks. We want to assign tasks to machines in such an order that we finish all the tasks as early as possible. By "online" we mean we cannot control the order in which the tasks arrive.

Input:

- k - Number of machines we have.
- $t_1, t_2, ..., t_n$ - n tasks, task i takes t_i amount of time to finish.

Output: A function S which assigns tasks to machines $S : \{t_1, t_2, ..., t_n\} \to \{1, 2, ..., k\}$.

Goal: To minimize $q(S)$ - The time in which the last task finishes.

We will show a greedy algorithm which gives a $\left(2 - \frac{1}{k}\right)$-approximation to the problem.

3.1.1.1 Algorithm

Algorithm 3.1 Parallel machine online scheduling $\left(2 - \frac{1}{k}\right)$-approximation algorithm

---

1. For each task i from 1 to n:
   (a) Send task i to the machine which is currently planned to finish first.

In other words, the algorithm sends the current task to the least occupied machine.

For example, given the following tasks:
$t_1 = 1, t_2 = \frac{1}{2}, t_3 = \frac{2}{3}, t_4 = 1$
and two machines ($k = 2$), the algorithm will do the following:

1. Task 1 is assigned to machine 1:

Machine 1	Machine 2
t_1	

2. Task 2 is assigned to machine 2:

Machine 1	Machine 2
t_1	t_2

3. Machine 2 is less occupied than machine 1 ($t_2 < t_1$), so task 3 is assigned to machine 2:

Machine 1	Machine 2
t_1	t_2, t_3

4. Machine 1 is less occupied than machine 2 ($t_1 < t_2 + t_3$), so task 4 is assigned to machine 1:

Machine 1	Machine 2
t_1, t_4	t_2, t_3

In this case, machine 1 will need $t_1 + t_4 = 2$ time to finish, and machine 2 will need $t_2 + t_3 = 1\frac{1}{6}$ time to finish. So $q(S) = 2$.

This is an approximating algorithm, so it doesn't promise the best result. And indeed there is a better solution:

Machine 1	Machine 2
t_1, t_2	t_3, t_4

, for which $q(S) = 1\frac{2}{3}$.

3.1.1.2 Proof of $\left(2 - \frac{1}{k}\right)$-approximation

We need to show that the algorithm in 3.1.1.1 is $\left(2 - \frac{1}{k}\right)$-approximating. In other words, if we denote the solution of the algorithm as S, and the uknown optimal solution as $S^\star$, then we need to show that $q(S) \leq \left(2 - \frac{1}{k}\right) q(S^\star)$.

Obviously we don't know what is $S^\star$, but we can determine some characteristics it must hold. We do this in the following two lemmas.

Lemma 2. *The time in which the last machine finishes in the optimal solution, is greater than the average finish time between all machines:* $q(S^\star) \geq \frac{1}{k} \sum_{i=1}^{n} t_i$.

Lemma 3. *The time in which the last machine finishes in the optimal solution, is greater than the time of the longest task (denoted by* t_{max}*):* $q(S^\star) \geq t_{max}$.

We won't prove these lemmas, as they are self evident. We now continue with the proof.

Let's consider the last decision our algorithm has made regarding the machine which finishes last. We denote the index of the task of this decision as l ($1 \leq l \leq n$), and the machine as j ($1 \leq j \leq k$). We also denote as F_j the time in which machine j finishes after the first $l - 1$ tasks have been assigned.

We also know that, by the definition of the algorithm, $F_j \leq \frac{1}{k}\sum_{i=1}^{l-1} t_i$ (because otherwise task l wouldn't be assigned to this machine).

Since j is the machine which finished last, we know that $q(S) = F_j + t_l$.

Thus we have $q(S) \leq \frac{1}{k}\sum_{i=1}^{l-1} t_i + t_l = \frac{1}{k}\sum_{i=1}^{l} t_i + \left(1-\frac{1}{k}\right) t_l \leq \frac{1}{k}\sum_{i=1}^{n} t_i + \left(1-\frac{1}{k}\right) t_{max} \leq q(S^\star) + \left(1-\frac{1}{k}\right) q(S^\star) = \left(2-\frac{1}{k}\right) q(S^\star)$. ■

3.1.2 Set Cover Problem

We have m subsets of $[n]$*: $A_i \subseteq [n]$ and $1 \leq i \leq m$. We also know that the sets A_i cover all of $[n]$: $\cup_{i=1}^{m} A_i = [n]$.

We want to find the minimal partial group of these sets that still covers all of $[n]$. In other words, we want to find $S \subseteq \{1,...,m\}$ such that $\cup_{i\in S} A_i = [n]$ and $q(S) = |S|$ is minimal.

3.1.2.1 $\ln(n)$-Approximating algorithm

We propose an iterative algorithm. In each iteration we denote by X the set of numbers in $[n]$ which are still not covered ($X \subseteq [n]$). We also denote by T_i the relevant part of set A_i ($T_i = A_i \cap X$).

Algorithm 3.2 Minimal set cover approximating algorithm

1. Initialize $S = \emptyset$
2. Denote by i the index in which $|T_i|$ is maximal.
3. Update:
   (a) $S = S \cup \{i\}$
   (b) $X = X \backslash T_i$
   (c) $\forall j = 1,...,m : T_j = T_j \backslash T_i$
4. Repeat steps 2,3 until $X = \emptyset$
5. Return S

3.1.2.2 Proof of $\ln(n)$-approximation

We want to prove that algorithm 3.2 gives a $\ln(n)$-approximation to the optimal solution. In other words, we want to show that $|S| \leq \ln(n)\,|S^\star|$.

We prove by examining the change in the size of X in each iteration of the algorithm.

We denote n_j the size of X after j iterations. Obvisouly $n_0 = n$ and if $n_j = 0$ then $|S| \leq j$.

*$[n] = \{1, 2, ..., n\}$

Let's examine the $j+1$ iteration of the algorithm (currently $|S| = j$). As denoted earlier, T_i is the chosen set in this iteration, which means $|T_i|$ is maximal. By our definitions, we have $n_{j+1} = |X \backslash T_i| = n_j - |T_i|$.

We use the following lemma:

Lemma 4. $|T_i| \geq \frac{|X|}{|S^\star|}$.

Proof. We need to show that $|T_i| \geq \frac{|X|}{|S^\star|}$, or equivalently $|X| \leq |S^\star| \cdot |T_i|$.

$|T_i|$ has the maximal size of all T's, so $\forall j \neq i : |T_i| \geq |T_j|$.

$S^\star$ is the optimal solution.

By the definition of X and the T's we can write $|X| = |\cup_{j \in S^\star} T_j|$.

So, $|X| = |\cup_{j \in S^\star} T_j| \leq \cup_{j \in S^\star} |T_j| \leq |S^\star| \cdot |T_i|$, as needed. □

We continue with the proof of the approximation.

We had $n_{j+1} = n_j - |T_i|$, so by the lemma, $n_{j+1} \leq n_j - \frac{|X|}{|S^\star|} = n_j - \frac{n_j}{|S^\star|} = n_j \left(1 - \frac{1}{|S^\star|}\right)$. This is a recursive formula from which we can conclude:

$n_j \leq n_{j-1} \left(1 - \frac{1}{|S^\star|}\right) \leq n_{j-2} \left(1 - \frac{1}{|S^\star|}\right)^2 \leq ... \leq n_0 \left(1 - \frac{1}{|S^\star|}\right)^j = n \left(1 - \frac{1}{|S^\star|}\right)^j$.

We will employ the following lemma (which will not be proven here):

Lemma 5. $(1+a)^b < e^{ab}$, *for all non zero* $a, b \in \mathbb{R}$.

We use the lemma and obtain: $n_j \leq n \left(1 - \frac{1}{|S^\star|}\right)^j \leq n e^{\frac{-j}{|S^\star|}}$.

Since n_j in an integer, then for $j = |S^\star| \ln(n)$ we obtain $n_j = 0$, which, as stated earlier, means that $|S| \leq |S^\star| \ln(n)$, thus completing the proof. ■

3.1.3 Max-Cut Problem

We have an undirected graph $G = (V, E)$.

A cut in the graph, $C = (A, B)$, is such that $A, B \subseteq V$ and $A \uplus B = V$*.

We denote with E_C the edges of the cut: $E_C = \{(i,j) \in E : i \in A, j \in B\}$.

Our goal is to find a cut C with maximum edges in the graph (maximal $|E_C|$).

*$\uplus$ is the symbol for disjoin union, i.e. $A \uplus B = V$ means $A \cup B = V$ and $A \cap B = \emptyset$.

3.1.3.1 2-approximating algorithm

Algorithm 3.3 Max-Cut 2-approximating algorithm

1. Initialize two sets: $A = V$, $B = \emptyset$.
2. For each vertex $i \in V$:
   (a) Denote by X the set to which i belongs (A or B).
   (b) Denote by Y the other set.
   (c) If i has more neighbors in X than in Y, move i to Y.
3. Repeat step 2 until no more changes occur.

At first it may seem the algorithm might never stop. But we notice that the quality of any solution ($|E_C|$) is bounded by the number of edges in the graph ($|E_C| \leq |E|$). In each iteration of step 2, we move a vertex to the set in which it has less neighbors (or leave it, if it's already there), which means that $|E_C|$ either increases or stays the same, but it never decreases. But because $|E_C|$ is bounded, then the algorithm must eventually stop.

3.1.3.2 Proof of 2-approximation

We denote by d_v the degree of vertex v, and by v_C the number of edges in the cut that touch vertex v.

Every edge in the cut touches two vertices, so $|E_C| = \frac{1}{2}\sum_{v\in V} v_C$.

By the operation of the algorithm, it makes sure that for every vertex, at least half of its neighbors are in the other set, which means that at least half of the edges that touch this vetex are in the cut: $\forall v \in V : v_C \geq \frac{1}{2}d_v$.

So: $|E_C| = \frac{1}{2}\sum_{v\in V} v_C \geq \frac{1}{4}\sum_{v\in V} d_v$.

$\sum_{v\in V} d_v$ is exactly twice the edges in the graph (We count the number of edges for every vertex, but every edge touches two vertices): $\sum_{v\in V} d_v = 2\,|E|$.

Obviously, the optimal solution (denoted $C^\star$) doesn't have more edges in the cut than there are edges in the graph: $|E_{C^\star}| \leq |E|$

So now we have $|E_C| \geq \frac{1}{4}2\,|E| = \frac{1}{2}\,|E| \geq \frac{1}{2}\,|E_{C^\star}|$, which completes the proof. ■

3.1.4 Max-K-Cut

As in the *Max-Cut* problem in 3.1.3, here also we have an undirected graph $G = (V, E)$. Our goal here is to find a partition of V into k disjoint sets $V_1, V_2, ..., V_k$, such that the sum of the number of edges in the cuts (i.e. between all pairs of V_i, V_j) is maximized.

We notice that the *Max-Cut* problem is a private case of the current problem with $k = 2$.

3.1.4.1 $\left(1-\frac{1}{k}\right)$-approximating algorithm

This algorithm is very similar to algorithm 3.3. Here, too, in each iteration we move the current vertex into the set in which most of it's neighbors will be from the other sets.

We denote by d_v the degree of vertex v, by N_v^{in} the number of neighbors v has inside it's current set, and by N_v^{out} the number of neighbors v has outside of it's current set.

Algorithm 3.4 Max-K-Cut $\left(1-\frac{1}{k}\right)$-approximating algorithm

---

1. Partition V into k disjoint sets $V_1, V_2, ..., V_k$ (randomly).
2. For each vertex $i \in V$:
   (a) If $N_v^{in} > \frac{d_v}{k}$, move v from it's current set V_i to another set V_j, such that the new N_v^{out} is maximal.
3. Repeat step 2 until no more changes occur.

From the same considerations of algorithm 3.3, this algorithm must also eventually stop.

3.1.4.2 Proof of $\left(1-\frac{1}{k}\right)$-approximation

For a solution S we denote by $|E_S|$ the number of edges in the cuts defined by S. If S is the solution of our algorithm, and $S^\star$ is the optimal solution, we need to show that $|E_S| \geq \left(1-\frac{1}{k}\right)|E_{S^\star}|$.

We notice the following:

- $|E_S| = \frac{1}{2}\sum_{v\in V} N_v^{out}$
- $\forall v \in V : d_v = N_v^{out} + N_v^{in}$.
- $|E| = \frac{1}{2}\sum_{v\in V} d_v$.

After the algorithm finishes its operation, we know that $\forall v \in V : N_v^{in} < \frac{d_v}{k}$.

Thus we conclude $\forall v \in V : N_v^{out} \geq \frac{k-1}{k} d_v$.

So now we can write: $|E_S| = \frac{1}{2}\sum_{v\in V} N_v^{out} \geq \frac{1}{2}\frac{k-1}{k}\sum_{v\in V} d_v = \frac{k-1}{k}|E|$.

We also know that in any solution, as well as the optimal solution, the number of edges in the cuts cannot exeed the number of edges in the graph, so: $|E_{S^\star}| \leq |E|$.

So finally we have: $|E_S| \geq \frac{k-1}{k}|E| \geq \frac{k-1}{k}|E_{S^\star}| = \left(1-\frac{1}{k}\right)|E_{S^\star}|$, thus completing the proof. ■

3.1.5 Vertex Cover

We have a graph of m vertices and n edges: $G = (V, E)$.

We want to find a subset $S \subseteq V$ such that every edge has at least one vertex in that set. In other words $\forall e = (i, j) \in E : i \in S \, or \, j \in S$.

Our goal is to find such a set with minimal size (minimize $q(S) = |S|$).

We notice this is a special case of the *Set Cover Problem* presented in 3.1.2, because if we number the edges of the graph $1, ..., n$, we can define the set A_i to be the vertices of edge i. So we can solve this problem with the same $\ln(n)$-approximating algorithm.

Here, however, we will show a 2-approximating algorithm.

First we will see several examples.

Example 6. The complete graph with m vertices (K_m). In this case $q(S^\star) = m - 1$, because if we leave out 2 vertices, the edge which connects them will not be covered.

Example 7. The bipartite complete graph $K_{a,b}$. If $a \leq b$ then obviously $q(S^\star) = a$.

Example 8. A matching of size k - a graph with k edges where each vertex has degree of excatly 1 (connected to exactly 1 edge). Obviously $q(S^\star) = k$.

Corollary 9. *If a graph* ***contains*** *a matching of size k, then $q(S^\star) \geq k$.*

3.1.5.1 2-approximating algorithm

Algorithm 3.5 Vertex cover 2-approximating algorithm

1. Initialize: $S = \emptyset$, $X = E$.
2. While $X \neq \emptyset$:
   (a) Choose some edge $e = (i, j) \in X$.
   (b) Add to S vertices i, j: $S = S \cup \{i, j\}$.
   (c) Remove from X all edges which are covered by i or j: $X = X \setminus \{All \, edges \, that \, touch \, i \, or \, j\}$.
3. Return S.

3.1.5.2 Proof of 2-approximation

We denote the solution of the algorithm with S, and the optimal solution with $S^\star$.

We want to show that $q(S) \leq 2q(S^\star)$ ($|S| \leq 2|S^\star|$).

By the structure of the algorithm, $|S|$ is even (we add two vertices each iteration), so we can write $|S| = 2k$ for some k. We notice that all the edges

the algorithm choses in step 2a have no vertices in common (becuase of step 2c). In other words, these edges make a matching of k edges. By corollary 9 we conclude $|S^\star| \geq k$.

Finally we have $|S| = 2k \leq 2|S^\star|$, thus completing the proof. ■

3.1.6 Metric Travelling Salesman

We have a **complete** undirected graph $G = (V, E)$, and a **metric** weight function $w : E \to \mathbb{R}^+$. Since w is metric, is holds the triangle inequallity: $\forall i, j, k \in V : w(i, k) \leq w(i, j) + w(j, k)$.

Our goal is to find a simple cycle C in the graph ($C = (e_1, e_2, ..., e_k)$) which touches **every** vertex exactly once, and has a minimal weight ($w(C) = \sum_{e \in C} w(e)$ is minimal).

3.1.6.1 2-approximating algorithm

Algorithm 3.6 Metric Travelling Salesman 2-approximating algorithm

1. Initialize $C = \emptyset$.
2. Find a minimal spanning tree (MST) of the graph, and denote it with T.
3. Double the edges in T to form a cycle which touches all the vertices, and denote it with T'.
4. Traverse the edges of T' starting from some arbitrary vertex:
   (a) Denote the current vertex with u.
   (b) Denote the next **unvisited** vertex with v.
   (c) Add the edge (u, v) to C: $C = C \cup \{(u, v)\}$.
   (d) Continue the next iteration from v.
5. Return C.

Proof of correctness We need to prove that the returned C is indeed a simple cycle which touches every vertex exactly once.

We notice that T' is a cycle (not simple) which touches every vertex. In step 4, by traversing **every** edge of T', we visit **every** vertex in the graph. But because we only visit the **unvisited** vertices, we know that we touch every vertex **once**. Because G is a **complete** graph, there is always an edge between the vertices u and v in step 4, thus making C a simple cycle. ■

3.1.6.2 Proof of 2-approximation

We denote by C the solution of algorithm 3.6, and by $C^\star$ the optimal solution. We need to prove that $w(C) \leq 2w(C^\star)$.

Claim 10. Let T_{MST} be a minimal spanning tree of G. Then $w(T_{MST}) \leq w(C^{\star})$.

Proof. Suppose we have $C^{\star}$. We remove one of it's edges. Now we have a spanning tree T. Removing an edge from $C^{\star}$ decreases it's weight by some amount. Thus we have $w(T) \leq w(C^{\star})$. From the minimality of T_{MST} we have $w(T_{MST}) \leq w(T)$. So finally we have $w(T_{MST}) \leq w(C^{\star})$. ■ □

In step 4 of the algorithm, from the current vertex u we connect an edge to the next **unvisited** vertex v, so we skip all the visited vertices between them (by the order defined by $T^{'}$). So by the triangle inequiallity (remember, w is metric), we conclude: $w(C) \leq w\left(T^{'}\right)$.

$T^{'}$ is formed by doubling the edges of T, so $w\left(T^{'}\right) = 2w(T)$.

T is a minimal spanning tree, so by claim 10 we have $w(T) \leq w(C^{\star})$.

Finally we can write $w(C) \leq w\left(T^{'}\right) = 2w(T) \leq 2w(C^{\star})$. Thus completing the proof. ■

3.1.6.3 1.5-approximating algorithm (by Christofides)

Algorithm 3.7 Metric Travelling Salesman 1.5-approximating algorithm (by Christofides)

1. Initialize $C = \emptyset$.
2. Find a minimal spanning tree for G and denote it with T.
3. Denotations:
   (a) A - The vertices with odd degree in T.
   (b) G_A - The induced graph of A by G.
4. Find a minimal perfect matching M^* in G_A.
5. Let C' be the edges in M^* combined with the edges in T (forming a multigraph).
6. Find an Euler cycle in C' and denote it with C_e.
7. Traverse the edges of C_e starting from some arbitrary vertex:
   (a) Denote the current vertex with u.
   (b) Denote the next **unvisited** vertex with v.
   (c) Add the edge (u, v) to C: $C = C \cup \{(u, v)\}$.
   (d) Continue the next iteration from v.
8. Return C.

Notes:

1. In step 3b, G_A is a complete graph, because it's induced on a complete graph.
2. In step 6, an *Euler cycle* is a cycle (simple or not) which touches every vertex exactly once.

In order to use the algorithm, we need the following claims:

Claim 11. Every graph G has an even number of vertices with odd degree.

Proof. Let d_v denote the degree of vertex v. In every graph $G = (V, E)$, $\sum_{v \in V} d_v = 2|E|$. If the number of vertices with odd d_v is odd, then $\sum_{v \in V} d_v$ will be odd. But this is impossible. ■ □

Claim 12. Every connected graph in which for every vertex v, d_v is even, there is an Euler cycle.

Claim 13. In a graph with even number of vertices, it is possible to find a minimal perfect matching in polynomial runtime (Using Edmond's *Path, Trees and Flowers* algorithm, or Vazirani's algorithm).

Claim 14. In C' there is a cycle.

Proof. We will show that every vertex i in C' has an even degree, so it must contain a cycle. If d_i is even in T, then adding M^* does not affect i's degree, because M^* is a matching of vertices with odd degree. If d_i is odd in T, then adding M^* adds 1 to d_i, because M^* is a matching, and as such connects only one edge to each vertex. □

Claim 15. In C' there is an Euler cycle.

Proof. We have shown that the degree of every vertex in C' is even, so we can use claim 12. □

In step 6 we obtained an Euler cycle, which we transform to a simple cycle which touches every vertex, just as we did in algorithm 3.6.

3.1.6.4 Proof of 1.5-approximation

We denote by C the solution of algorithm 3.7 and by $C^\star$ the optimal solution.

We want to prove $w(C) \leq 1.5w(C^\star)$.

Obviously $w(C) \leq w(T) + w(M^*)$.

So we will devide the proof into two parts.

First we will show $w(T) \leq w(C^\star)$, and then $w(M^*) \leq 0.5w(C^\star)$.

From claim 10 we conclude $w(T) \leq w(C^\star)$.

Let's denote with $C^\star_A$ the optimal solution for the problem on G_A. By the triangle inequality, we can deduce that an optimal solution which touches only part of the vertices, is better (has less weight) than an optimal solution that touches all of the vertices. Thus we conclude $w(C^\star_A) \leq w(C^\star)$.

The set A contains an even number of vertices, and $C^\star_A$ is a cycle which touches all these vertices. This means that in $C^\star_A$ there are two perfect mathces (alternating edges of the cycle). We denote them M^1_A and M^2_A, and assume, WLOG, M^1_A has the smaller weight ($w\left(M^1_A\right) \leq w\left(M^2_A\right)$). Both M^1_A and M^2_A are also perfect matches in G_A, and M^* is defnied to be the **minimal** perfect matching M^* in G_A, thus $w(M^*) \leq w\left(M^1_A\right)$.

Also, from the definition of M^1_A and M^2_A: $w(C^\star_A) = w\left(M^1_A\right) + w\left(M^2_A\right)$, and from the minimality of M^1_A: $w\left(M^1_A\right) \leq \frac{1}{2}w(C^\star_A)$.

From this we conclude $w(M^*) \leq w\left(M^1_A\right) \leq \frac{1}{2}w(C^\star_A) \leq \frac{1}{2}w(C^\star)$.

Finally we have $w(C) \leq w(T) + w(M^*) \leq w(C^\star) + \frac{1}{2}w(C^\star) = 1.5w(C^\star)$, as needed to complete the proof. ■

3.1.7 3-SAT (Satisfiability)

Before we define the problem, some definitions and denotations:

- Boolean variable: $x \in \{\mathbb{T}, \mathbb{F}\}$.
- Literal: either a variable x or it's negation $\neg x$.
- Clause: A disjunction of literals (such as $x_1 \vee \neg x_2 \vee x_3$).
- Formula: A conjunction of clauses (such as $(x_1 \vee \neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_4 \vee x_5)$). Such a formula is said to be in *conjunctive normal form* (CNF).
- A *k-CNF* formula is a CNF formula in which each clause contains k literals.

In this problem, our input is a 3-CNF formula with m clauses. Our goal is to find an assignment for the variables in the formula such that the most clauses are satisfied (i.e. evaluated valuated as $\mathbb{T}$).

Let X be an assignment of the variables, we denote $q(X)$ as the number of satisfied clauses.

Assumptions:

1. No clause contains the same literal more than once.
2. No clause contains a variable and it's negation.

3.1.7.1 2-approximating algorithm

Algorithm 3.8 3-SAT 2-approximating algorithm

1. Denote the assignment in which all variables are assigned $\mathbb{T}$ with $X_{\mathbb{T}}$.
2. Denote the assignment in which all variables are assigned $\mathbb{F}$ with $X_{\mathbb{F}}$.
3. If $q(X_{\mathbb{T}}) > q(X_{\mathbb{F}})$, return $X_{\mathbb{T}}$.
4. Else, return $X_{\mathbb{F}}$.

3.1.7.2 Proof of 2-approximation

Denote with m the number of clauses. Let S be the solution of the algorithm and $S^\star$ the optimal solution.

If $X_{\mathbb{T}}$ (as defined in the algorithm) does not satisfy a clause C_i, then it must be satisfied by $X_{\mathbb{F}}$ (because a clause is a disjunction of literals). Thus $q(X_{\mathbb{T}}) + q(X_{\mathbb{F}}) = m$.

We also know that the optimal solution cannot satisfy more than m clauses, so $q(S^\star) \leq m$.

Also, we know that $S = \max\{X_{\mathbb{T}}, X_{\mathbb{F}}\}$, so $q(S) \geq \frac{1}{2}m$.

Finally we have $q(S) \geq \frac{1}{2}m \geq \frac{1}{2}q(S^\star)$, thus completing the proof. ■

Chapter 4

Probabilistic Algorithms

4.1 Examples

4.1.1 Max-Lin-2

This is a problem of solving a linear system of equations over a finite field of two elements* ($F_2 = \{0, 1\}$).

The input is a system of m equations with n variables over the field F_2.

Our goal is to find an assignment for the n variables, such that as much of the equations are satisfied.

In other words, if we our solution is S and we define $q(S) = number\,of\,satisfied\,equations$, then we would like to maximze $q(S)$.

Notice that if there is an assignment which satisfies **all** the equations, then we already know how to find it by employing one of the methods of solving a linear system of equations. But the more interesting problem in our case, is when not all equations can be satisfied by any assignment. In which case we would like to find the assignment which maximizes the number of satisfied equations.

Example 16. The system of equations $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ is solved with $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

Example 17. The system of equations $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$ is

*$F_2 = \{0, 1\}$ is a finite field (also called *Galois field*), in which the results of arithmetic operations are taken modulus 2. For example: $1 + 1 = 0$.

solved with $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$. Notice that in this case we have more equations than variables.

Example 18. The system of equations $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$ **cannot** be solved. However, $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ satisfies the top 3 equations, and only the fourth equation is not satisfied.

4.1.1.1 2-approximating probabilistic algorithm

We will now show an algorithm which yields a 2-approximating for the expectancy of the solutions. In other word, if the best solution, $S^\star$, satisfies $q(S^\star)$ equations, then this algorithm finds an assignment, S, such that if we consider the expactency of these assignments, at least $\frac{1}{2}q(S^\star)$ equations are satisfied ($\mathbb{E}[S] \geq \frac{1}{2}q(S^\star)$).

We denote the n variables with $x_1, x_2, ..., x_n$.

Algorithm 4.1 Max-Lin-2 2-expectancy-approximation probabilistic algorithm

1. For $i = 1, ..., n$:
   (a) Assign $x_i = 0$ or $x_i = 1$ with probability of 0.5 (toss a coin).

4.1.1.2 Proof of 2-approximation of the expectancy

Notice that $q(S^\star) \leq m$, so it's enough to show $\mathbb{E}[S] \geq \frac{m}{2}$.

We define the following probability space:

$\Omega = \{\omega = \{\omega_1, \omega_2, ..., \omega_n\} : \omega_i \in \{0, 1\}\}$

Each $\omega \in \Omega$ is an assignment for $x_1, x_2, ..., x_n$.

We define the probability function $P(\omega) = \frac{1}{2^n}$ (because in each ω we have n elements with probability $\frac{1}{2}$).

We also define a random variable $X(w)$ as the number of equations ω satisfies ($X(\omega) = |\{i : (A\omega)_i = b_i\}|$).

The expectancy of X is the average number of equations satisfied by a random assignment.

We will show that $\mathbb{E}[X] = \frac{m}{2}$.

Let's define m new random variables $X_1(\omega), X_2(\omega), ..., X_m(\omega)$, where $X_i(\omega)$ indicates if equation i is satisfied: $X_i(\omega) = \begin{cases} 1 & (A\omega)_i = b_i \\ 0 & (A\omega)_i \neq b_i \end{cases}$.

By this definition, we have $X(w) = \sum_{i=1}^{m} X_i(\omega)$, and by the linearity of the expectation $\mathbb{E}[X] = \sum_{i=1}^{m} \mathbb{E}[X_i]$.

We will now see that $\forall i = 1, ..., m : \mathbb{E}[X_i] = \frac{1}{2}$, and thus $\mathbb{E}[X] = \frac{m}{2}$.

$$\mathbb{E}[X_i] = \sum_{\omega\in\Omega} P(\omega) \cdot X_i(\omega) = \sum_{\omega\in\Omega:X_i(\omega)=1} P(\omega) = \underbrace{\frac{1}{2^n}\left(\sum_{\omega\in\Omega:X_i(\omega)=1} 1\right)}_{*}.$$

The expression marked with $*$ is the number of assignments which can satisfy equation i. From linear algebra we know that a linear equation of n variables has 2^{n-1} legal assignments (over the field F_2)*.

So finally we have $\mathbb{E}[X_i] = \frac{2^{n-1}}{2^n} = \frac{1}{2}$, thus copmleting the proof. ■

4.1.1.3 Further discussion

Probability of a random assignment We can also determine the probability in which a certain amount of equations are satisfied by an assignment ω.

For example, let's show $P(\omega : X(\omega) \geq 0.4m) \geq \frac{1}{6}$, or in words - we will show that the probability that an assignment ω satisfies at least 40% of the equations is larger than $\frac{1}{6}$.

Let's detone with B the event $B = \{\omega : X(\omega) \geq 0.4m\}$. We want to prove $P(B) \geq \frac{1}{6}$. We will prove by negation.

Suppose $P(B) < \frac{1}{6}$.

$\mathbb{E}[X] = \sum_{\omega\in\Omega} P(\omega) \cdot X(\omega) = \sum_{\omega\in B} P(\omega) \cdot X(\omega) + \sum_{\omega\notin B} P(\omega) \cdot X(\omega)$

Any assignment cannot satisfy more than m equations (obviously, since there are only m equations), so $\sum_{\omega\in B} P(\omega) \cdot X(\omega) \leq \sum_{\omega\in B} P(\omega) \cdot m$.

Also, by the definition of B: $\sum_{\omega\notin B} P(\omega) \cdot X(\omega) < \sum_{\omega\notin B} P(\omega) \cdot 0.4m$.

So now we have $\mathbb{E}[X] < \sum_{\omega\in B} P(\omega) \cdot m + \sum_{\omega\notin B} P(\omega) \cdot 0.4m = mP(B) + 0.4mP(B^c)$, where B^c is the complementary event of B.

$P(B^c) = 1 - P(B)$, so $\mathbb{E}[X] < mP(B) + 0.4m(1 - P(B)) = 0.6mP(B) + 0.4m$.

But we assumed $P(B) < \frac{1}{6}$, thus $\mathbb{E}[X] < \frac{1}{6}0.6m + 0.4m = 0.5m$.

Finally we have $\mathbb{E}[X] < 0.5m$. But we have already shown $\mathbb{E}[X] = 0.5m$, thus we have reached a contradiction, and we can conclude $P(B) \geq \frac{1}{6}$. ■

An improved algorithm By 4.1.1.3 we can formulate an improved algorithm which yields, by very good probability (specifically $1 - \frac{1}{e^{100}} \approx 99.999\%$), an assignment which satisfies at least 40% of the equations.

Notice that $\frac{1}{0.4} = 2.5$, so this is a high-probability 2.5-approximating algorithm. As opposed to the previous algorithm, here we don't say anything about the expectancy of the solutions the algorithms yields, but we say a much stronger statement about a specific solution. Altough 2.5-approximation is not

*Consider an equation of the form $a_1x_1 + a_2x_2 + ... + a_nx_n = b$. In F_2 each $x_i \in \{0, 1\}$. We can set $n - 1$ variables with either 1 or 0, by which the n'th variable is determined. This gives 2^{n-1} combinations.

as good as 2-approximation (40% vs. 50%), here we have, with high-probability, a specific solution which satisfies at least 40% of the equations.

Algorithm 4.2 Max-Lin-2 2.5-approximation high-probability algorithm

1. Choose 600 random assignments $\omega_1, \omega_2, ..., \omega_{600}$.
2. Return the assignment which satisfies the most equations.

By 4.1.1.3 we know that assignment ω_i satisfies 40% of the equations with probability of $\frac{1}{6}$. The probability that all 600 chosen assignments will fail to do this is $\left(1-\frac{1}{6}\right)^{600}$. By lemma 5, we have $\left(1-\frac{1}{6}\right)^{600} < e^{-100}$.

So the probability that at least one assignment will satisfy 40% of the equations is at least $1-\frac{1}{e^{100}}$. ■

4.1.2 3-SAT (Satisfiability)

See the definition of the problem in 3.1.7.

4.1.2.1 Probabilistic $\frac{7}{8}$-approximating algorithm

We show a probabilistic algorithm which gives a $\frac{7}{8}$-approximation with probability larger than $1-\frac{1}{e^k}$ (with k as a parameter of the algorithm).

Algorithm 4.3 3-SAT probabilistic $\frac{7}{8}$-approximating algorithm

1. Randomly assign $\mathbb{T}$ or $\mathbb{F}$ to each variable x_i (with probability 0.5).
2. If $q(X) \geq \frac{7}{8}m$, finish and return X.
3. Repeat steps 1-2 at most $k(m+1)$ times.

4.1.2.2 Proof

We define *success* of the algorithm as the success in yielding a solution, X, which satisfies at least $\frac{7}{8}m$ clauses. In other words, we succeed only if $q(X) \geq \frac{7}{8}m$, otherwise we *fail*.

We will prove in two parts. First we will show that the probability for success of steps 1-2 (which will be reffered to as the *basic* steps) is $P(success\,basic) \geq \frac{1}{m+1}$. Then we will show that the probability for success of the entire algorithm (all 3 steps) is $P(success) \geq 1-\frac{1}{e^k}$.

Part 1: We need to prove $P(success\,basic) \geq \frac{1}{m+1}$.

We define the probability space as all the tuples of $\mathbb{T}$ and $\mathbb{F}$ of size n: $\Omega = \{\mathbb{T}, \mathbb{F}\}^n$.

Obviously there are 2^n such tuples, so for some $\omega \in \Omega$ we have $P(\omega) = \frac{1}{2^n}$.
We define two random variables:

- $X(\omega)$ - The amount of satisfied clauses by ω. Thus $P(success\,basic) = P\left(X \geq \frac{7}{8}m\right)$.
- $Y(\omega)$ - The amount of unsatisfied clauses by ω. Thus $P(failure\,basic) = P\left(Y > \frac{1}{8}m\right)$.

Instead of proving $P\left(X \geq \frac{7}{8}m\right) \geq \frac{1}{m+1}$, we can equivalently prove $P\left(Y > \frac{1}{8}m\right) < 1 - \frac{1}{m+1} = \frac{m}{m+1} = \frac{1}{1+\frac{1}{m}}$.

We define m new random variables:

$$\forall i = 1...m : Y_i(\omega) = \begin{cases} 1 & clause\,i\,is\,satisfied\,by\,\omega \\ 0 & clause\,i\,is\,unsatisfied\,by\,\omega \end{cases}.$$

Obviously $Y(\omega) = \sum_{i=1}^{m} Y_i(\omega)$, so by the linearity of the expectation $\mathbb{E}[Y] = \sum_{i=1}^{m} \mathbb{E}[Y_i]$.

$\mathbb{E}[Y_i] = P(Y_i = 1) \cdot 1 + P(Y_i = 1) \cdot 0 = P(Y_i = 1)$

$Y_i = 1$ means that clause i is satisfied. In 3-SAT, each clause contains 3 literals, each of them can evaluate to $\mathbb{T}$ or $\mathbb{F}$ by the assignment. Thus the probability for a clause to be satisfied is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$. Thus $\mathbb{E}[Y_i] = P(Y_i = 1) = \frac{1}{8}$.

And now $\mathbb{E}[Y] = \sum_{i=1}^{m} \mathbb{E}[Y_i] = \frac{m}{8}$.

We now recall Markov's inequality:

For a non-negative random variable, Z, and some constant $c > 1$: $P(Z \geq c \cdot \mathbb{E}[Z]) \leq \frac{1}{c}$.

We use it on Y with $c = 1 + \frac{1}{m}$: $P\left(Y \geq \left(1 + \frac{1}{m}\right) \cdot \frac{m}{8}\right) \leq \frac{1}{1+\frac{1}{m}}$.

Both Y and m are integers, so $P\left(Y \geq \left(1 + \frac{1}{m}\right) \cdot \frac{m}{8}\right) = P\left(Y \geq \frac{m}{8} + \frac{1}{8}\right) = P\left(Y > \frac{m}{8}\right)$.

And finally we have $P(failure\,basic) = P\left(Y > \frac{m}{8}\right) \leq \frac{1}{1+\frac{1}{m}}$, as needed for this part of the proof. ■

Part 2: We need to prove $P(success) \geq 1 - \frac{1}{e^k}$.

Equivalently we can also prove $P(failure) < \frac{1}{e^k}$.

Failure in the entire algorithm is failure in all $k(m+1)$ attempts.

$P(failure) = \prod_{i=1}^{k(m+1)} P(failure\,in\,the\,i'th\,attempt)$.

In part 1 we already proved the failure in one attempt is $P(failure\,basic) \leq \frac{1}{1+\frac{1}{m}} = 1 - \frac{1}{m+1}$, thus:

$$P(failure) \leq \prod_{i=1}^{k(m+1)} \left(1 - \frac{1}{m+1}\right) = \left(1 - \frac{1}{m+1}\right)^{k(m+1)} = \left(\left(1 - \frac{1}{m+1}\right)^{(m+1)}\right)^{k} < \left(\frac{1}{e}\right)^{k} = \frac{1}{e^k},$$

thus completing the proof. ■

Chapter 5

Flow Networks

5.1 Motivation

5.1.1 Traffic

Let's think of traffic. We have junctions, which are connected by roads; and we have cars which drive in the roads. Let's take a look at the simple example depiceted in figure 5.1. In *Road A* there are two lanes, but in *Road B* there is only one lane. Obviously, if we use both lanes of *Road A*, very quickly there will be a traffic jam; because *Road B* can't contain the amount of traffic arriving from *Road A*. In order to avoid a traffic jam, we would have to limit the use of *Road A* to just one lane.

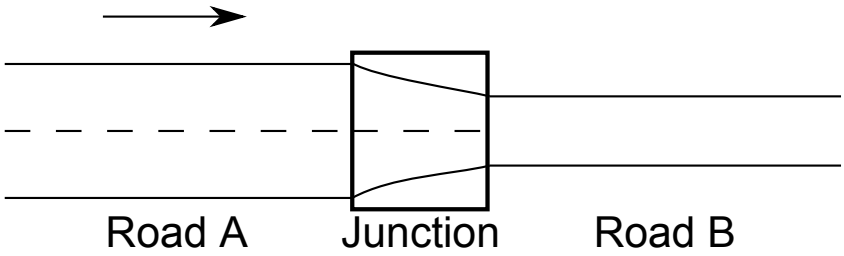
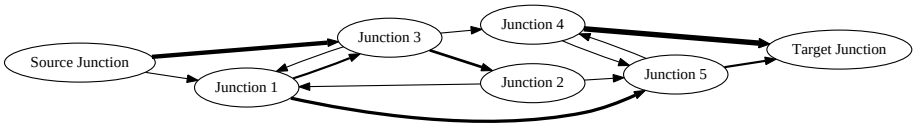


Figure 5.1: Traffic in a simple junction

Now let's look at it from the point of view of the junction. From the left, there is incoming traffic from both lanes of *Road A*, and from the right there is outgoing traffic to the single lane of *Road B*. If *Road B* can't remove traffic from the junction at the same rate that traffic is arriving to the junction from *Road A*, then the junction would become congested.

This simple analysis on this simple problem brought us to the conclusion that if we want to avoid traffic, we can use only one of the lanes of *Road A*. But real world problems are usually much more complex. Consider the situation

in which we have several junctions, each junction has several incoming and outgoing roads, and each road can have one or more lanes (figure 5.2). How would we find the best usage of each road such that none of the junctions would be congested?



The width of the road indicates it's capacity (number of lanes).

Figure 5.2: Traffic on many junctions and roads.

Chapter 6

Fast Fourier Transform

Index

List of Figures